Rechnerarchitektur Zusammenfassung

Maximilian Wolf

1. Oktober 2025

Inhaltsverzeichnis

1 Grundlagen

Definition 1.1: Rechnerarchitektur

Die Rechnerarchitektur beschreibt Eigenschaften und Verhalten eines Computers: Sie wird durch Befehle, deren Formate, Register, Speicherorganisation und arithmetische Operationen bestimmt. Das Verhalten ergibt sich aus Aufbau und Ablaufsteuerung. Architektur dient dazu, Rechner einheitlich zu klassifizieren, sodass Programmierer eine abstrakte Beschreibung erhalten und Ingenieure eine Spezifikation für den Aufbau.

1.1 Endlicher Automat

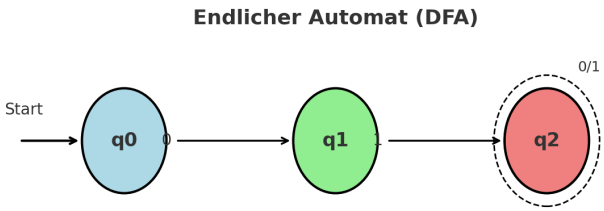

Abbildung 1: Aufbau eines endlichen Automaten

- **Zustandsmenge** $Z := \{Z_0, \cdots, Z_n\}$
- **Eingabealphabet** Σ
- **Zustandsüberführungsfunktion** $\delta := S \times \Sigma \to Z'$
- **Anfangszustand** $z_0 \in Z$
- **Endzustände** $E \subseteq Z$
- **Technische Umsetzung** durch Programmable Logic Array (PLA)

1.2 Turing-Maschine

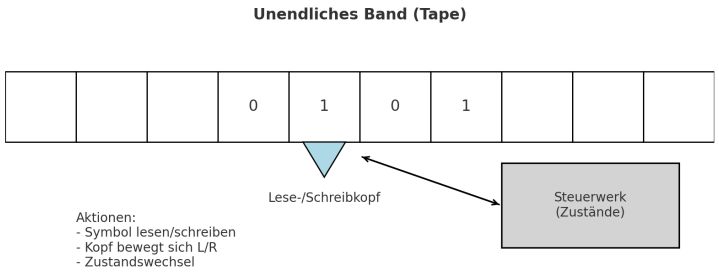

Abbildung 2: Aufbau einer Turing Maschine

- **Zustandsmenge** $Z := \{Z_0, \cdots, Z_n\}$
- **Eingabealphabet** $\Sigma := \{0, 1\}$
- **Bandalphabet** $\Gamma := \{0, 1, \square\} \supset \Sigma$
- **Zustandüberführungsfunktion** (Konfiguration): $\delta := Z \times \Gamma \times \{L, N, R\}$
- **Anfangszustand** $z_0 \in Z$
- **Endzustand** $Z_E \subseteq Z$

1.3 Verhalten vs. Struktur eines Rechners

1.3.1 Verhalten

- Funktion bzw. Arbeitsweise (dynamische Sicht)
- Wie werden Befehle abgearbeitet?
- Wie reagiert das System auf Ereignisse? (Eingaben, Signale, Interrupts)
- Beispiele
  - Ablauf einer Instruktion in der Pipeline
  - Speicherzugriffe
  - Ausführung einer Addition in der ALU
  - Steuerung durch Mikroprogramme

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ent is
  -- Entity definition
end entity;

architecture behavior of ent is
begin
  process(...)
  begin
    -- Process
  end process;
end architecture;
```

1.3.2 Struktur

- Aufbau des Systems (statische Ansicht)
- Komponenten und ihre Anordnung bzw. Verknüpfung
  - Prozessor, Speicher, Busse, I/O-Geräte
  - Steuerwerk, Rechenwerk, Registersatz, Cache-Hierarchie (4)
  - Topologie und Datenpfade

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ent is
-- Entity definition
end entity;

-- Entity instantiation
entity ent2 port map(...);

architecture structure of ent is
  p_out <= p_a AND p_b;
  ...
end architecture;
```

2 Befehlssatzarchitektur (ISA)

Definition 2.1: Befehlssatz

Moderne Computer sollen universell verwendbar, also programmierbar sein. Der Befehlssatz einer Maschine ist die Menge aller Befehle und der dazugehörigen Befehlsworte. Er ist die Schnittstelle zwischen dem Programm (der Software) und dem Rechner (der Hardware)

Definition 2.2: Pseudobefehl

Ist im Gegensatz zu einem Befehl nicht Teil der Instruktionsmenge (ISA), sondern wird vom Assembler in einen in der ISA enthaltenen Befehl übersetzt. Durch Pseudobefehle werden Programme einfacher und lesbarer.

2.1 Prinzip von Neumann

Definition 2.3: Prinzip von Neumann

1. **IF**: Befehl holen (Instruction Fetch)
2. **ID**: Befehl entschlüsseln (Instruction Decode)
3. **OF**: Eingabe holen (Operand Fetch)
4. **EX**: Befehl ausführen (Execute)
5. **OS**: Ausgabe schreiben (Operand Store)

2.2 n-Adress Maschine

Der Befehlssatz (Operation, Anzahl Operanden) eines Rechners wird durch die zu Grund legende Rechnerarchitektur maßgeblich bestimmt. Eine **Stapelmaschine** (0- bzw. 1-Adress Maschine) operiert auf einem Stapel (Stack), wohingegen eine **Registermaschine** nur auf Registern bzw. eine **Speichermaschine** nur auf dem Wahlfreien Speicher operiert.

2.2.1 Load-Store-Architektur

Load-Store Architektur bedeutet:

- Alle arithmetisch-logischen Operationen arbeiten nur auf Registern.

- Zugriff auf den Hauptspeicher erfolgt ausschließlich über spezielle Load- und Store-Befehle.

RISC-V ist ein typisches Beispiel für eine solche Architektur.

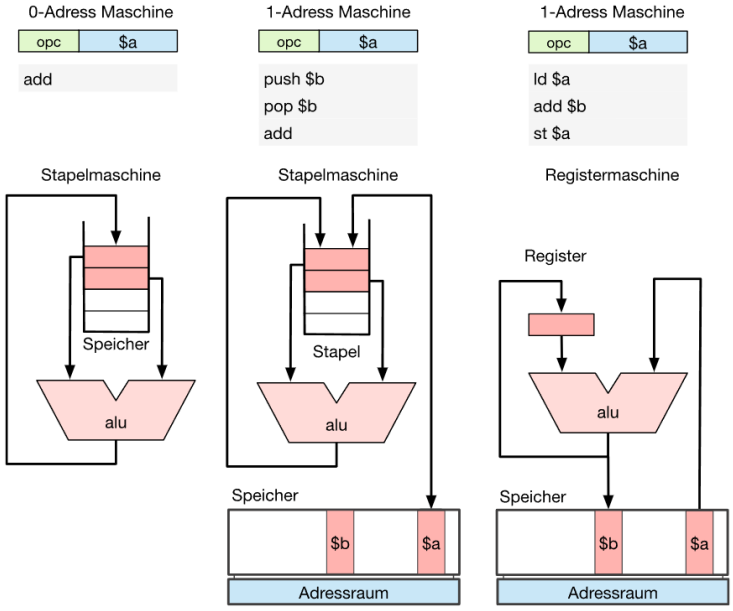

Abbildung 3: 0- und 1 Adress Maschine als Stapelmaschine und Registermaschine

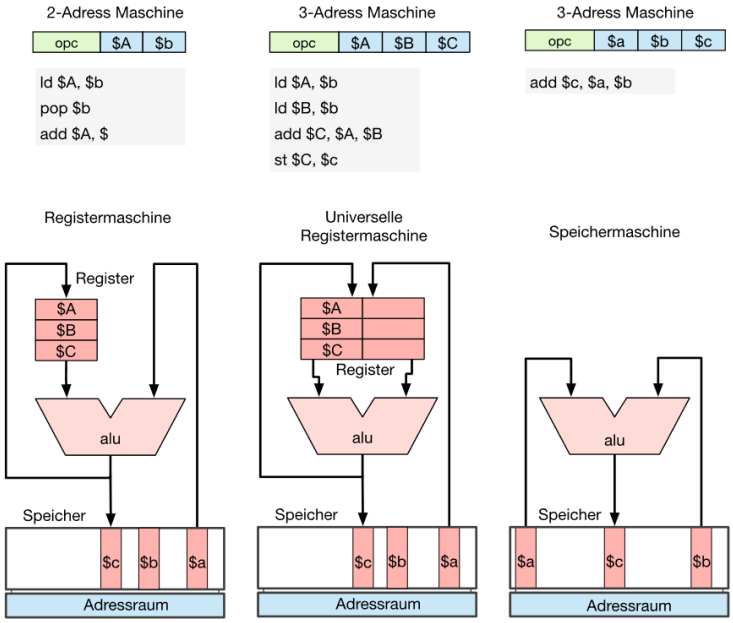

Abbildung 4: 2- und 3 Adress Maschine als Registermaschine und Speichermaschine

2.3 Befehlsgruppen

Definition 2.4: Opcode

Ein Opcode oder Operation-Code ist in einem Befehlssatz ein eindeutiges Bitmuster, das einen Befehl und dessen Operandenzugriffe eindeutig spezifiziert.

Einteilung der Befehle in

- **Arithmetische** Befehle: *add, sub, mul, div*
- **Logische** Befehle: *and, or, xor, (nand, nor)*

2.4 Adressierung

2.4.1 Inhärente Addressierung

- Opcode spezifiert Operandenzugriff
- z.B. push benötigt kein zusätzliches Argument für die Zieladresse

2.4.2 Implizite vs. unmittelbare Adressierung

- **Implizit**: Argument des Befehls wird direkt im Speicherwort abgelegt (**Immediates**)
- **Unmittelbar**: Argument des Befehls muss aus einer Adresse noch vor der Ausführung des Befehls geladen werden

2.4.3 Direkte / Absolute vs. Relative Adressierung

- **Direkt** bzw. **Absolut**: Adresse im Speicherwort wird als absolute Adresse im Speicher interpretiert, darin abgelegtes Datum entspricht dem Wert der Variablen
- **Relativ**: Adressierung erfolgt relativ zu einer *Basisadresse* + *Offset*
  - **PC-relativ**: Basisadresse wird durch aktuellen PC bestimmt
  - **Basisregister-relativ**: Basisadresse wird in einem angegebenen Register gespeichert

Registerdirekte Adressierung wird häufig zum Auffinden von Daten eingesetzt, wohingegen **Registerindirekte** Adressierung hauptsächlich für Sprünge verwendet wird.

2.4.4 Indirekte Adressierung

- An Speicherstelle wird nicht das zu verarbeitende Datum gespeichert, sondern eine weitere Adresse, an der dieses zu finden ist
- ermöglicht, in einem kleinen Adressraum Programme im Speicher kompakt zu halten.

2.4.5 Indizierte Adressierung

- Programme benötigen häufig Schleifen, um effizient nacheinander auf Elemente einer Datenstruktur (z.B. Liste, Array) zuzugreifen
- Die Laufvariable wird in einem speziellen **Indexregister** (auch X-, IX-, IDX-Register) gespeichert
- Die effektive Adresse ergibt sich aus der **Basisadresse** und dem Inhalt des **Indexregisters**

Beispiele für Befehle mit indizierter Adressierung in **RISC-V**

```
1 lw   x5, 12(x10) # R[5] = M[R[10] + 12]
2 sw   x5, 0(x10)  # M[R[10] + 0] = R[5]
3 lbu  x6, 3(x11)  # R[6] = M[R[11] + 3]
```

2.5 Orthogonaler Befehlssatz

Ein Befehlssatz ist **orthogonal**, wenn jede Operation alle Datentypen und Adressierungsarten implementiert. Da dies nicht immer sinnvoll ist, sind die Befehlssätze meistens **partiell orthogonal**

2.6 CISC vs. RISC

- **CISC** (Complex Instruction Set Computer): Unterstützung vieler Adressierungsarten & Befehlsformaten, die beliebig kombiniert werden
- **RISC** (Reduced Instruction Set Computer): Homogene Befehlsworte & beschränkte Anzahl an Adressierungsarten

Unterschiede in der Mikroarchitektur siehe 4.5

2.7 Befehlssatz von RISC-V

2.7.1 R-Type (Register-Register Operations)

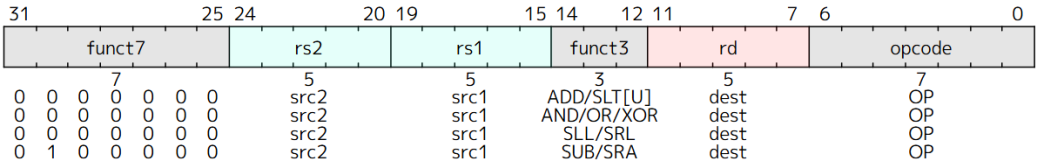

Abbildung 5: R-Type Befelsformat nach RV32I

Mnemonik	Name	Beschreibung (Verilog)
add	Add	R[rd] = R[rs1] + R[rs2]
sub	Subtract	R[rd] = R[rs1] - R[rs2]
and	AND	R[rd] = R[rs1] ∧ R[rs2]
or	OR	R[rd] = R[rs1] ∨ R[rs2]
xor	XOR	R[rd] = R[rs1] ⊕ R[rs2]
slt	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0
sltu	Set Less Than Unsigned	R[rd] = (unsigned R[rs1] < unsigned R[rs2]) ? 1 : 0
sll	Shift Left Logical	R[rd] = R[rs1] « R[rs2]
srl	Shift Right Logical	R[rd] = R[rs1] » R[rs2]
sra	Shift Right Arithmetic	R[rd] = R[rs1] »> R[rs2]

Tabelle 1: R-Type Instruktionen des RV32I Befehlssatzes

2.7.2 I-Type (Register-Immediate-Operations)

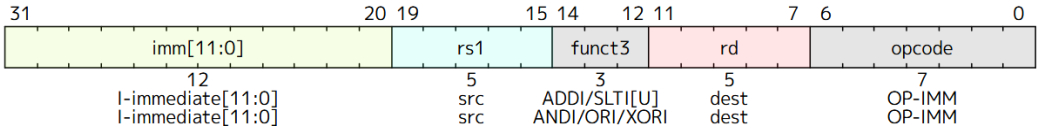

Abbildung 6: I-Type Befelsformat nach RV32I

Mnemonik	Name	Beschreibung (Verilog)
addi	Add Immediate	R[rd] = R[rs1] + imm
slti	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0
sltiu	Set Less Than Immediate Unsigned	R[rd] = (unsigned R[rs1] < unsigned imm) ? 1 : 0
andi	AND Immediate	R[rd] = R[rs1] ∧ imm
ori	OR Immediate	R[rd] = R[rs1] ∨ imm
xori	XOR Immediate	R[rd] = R[rs1] ⊕ imm
slli	Shift Left Logical Immediate	R[rd] = R[rs1] « shamt
srli	Shift Right Logical Immediate	R[rd] = R[rs1] » shamt
srai	Shift Right Arithmetic Immediate	R[rd] = R[rs1] »> shamt
lb	Load Byte	R[rd] = sign_extend(Mem[R[rs1] + imm][7:0])
lh	Load Halfword	R[rd] = sign_extend(Mem[R[rs1] + imm][15:0])
lw	Load Word	R[rd] = Mem[R[rs1] + imm][31:0]
lbu	Load Byte Unsigned	R[rd] = zero_extend(Mem[R[rs1] + imm][7:0])
lhu	Load Halfword Unsigned	R[rd] = zero_extend(Mem[R[rs1] + imm][15:0])
jalr	Jump And Link Register	R[rd] = PC + 4; PC = R[rs1] + imm

Tabelle 2: I-Type Instruktionen des RV32I Befehlssatzes

2.7.3 U-Type (Upper-Immediate-Operations)

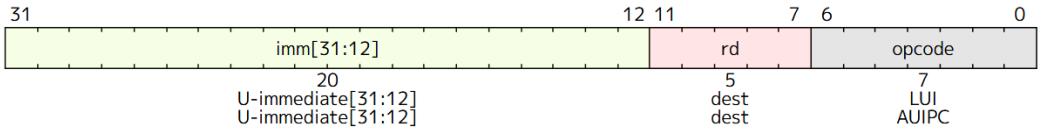

Abbildung 7: U-Type Befelsformat nach RV32I

Mnemonik	Name	Beschreibung (Verilog)
lui	Load Upper Immediate	R[rd] = imm « 12
auipc	Add Upper Immediate to PC	R[rd] = PC + (imm « 12)

Tabelle 3: U-Type Instruktionen des RV32I Befehlssatzes

2.7.4 B-Type (Branch-Operations)

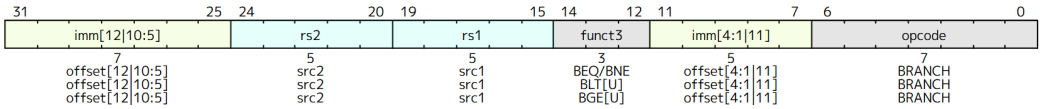

Abbildung 8: B-Type Befelsformat nach RV32I

Mnemonik	Name	Beschreibung (Verilog)
beq	Branch if Equal	if (R[rs1] == R[rs2]) PC = PC + (imm « 1)
bne	Branch if Not Equal	if (R[rs1] != R[rs2]) PC = PC + (imm « 1)
blt	Branch if Less Than	if (R[rs1] < R[rs2]) PC = PC + (imm « 1)
bge	Branch if Greater or Equal	if (R[rs1] >= R[rs2]) PC = PC + (imm « 1)
bltu	Branch if Less Than Unsigned	if ((unsigned R[rs1]) < (unsigned R[rs2])) PC = PC + (imm « 1)
bgeu	Branch if Greater or Equal Unsigned	if ((unsigned R[rs1]) >= (unsigned R[rs2])) PC = PC + (imm « 1)

Tabelle 4: B-Type Instruktionen des RV32I Befehlssatzes

2.7.5 S-Type (Store-Format)

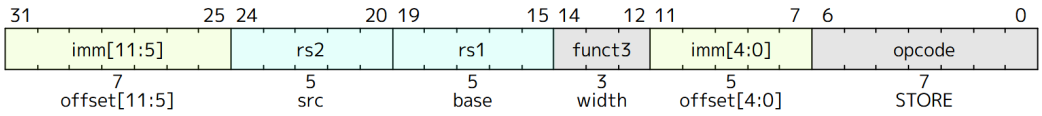

Abbildung 9: S-Type Befelsformat nach RV32I

Mnemonik	Name	Beschreibung (Verilog)
sb	Store Byte	Mem[R[rs1] + imm][7:0] = R[rs2][7:0]
sh	Store Halfword	Mem[R[rs1] + imm][15:0] = R[rs2][15:0]
sw	Store Word	Mem[R[rs1] + imm][31:0] = R[rs2][31:0]

Tabelle 5: S-Type Instruktionen des RV32I Befehlssatzes

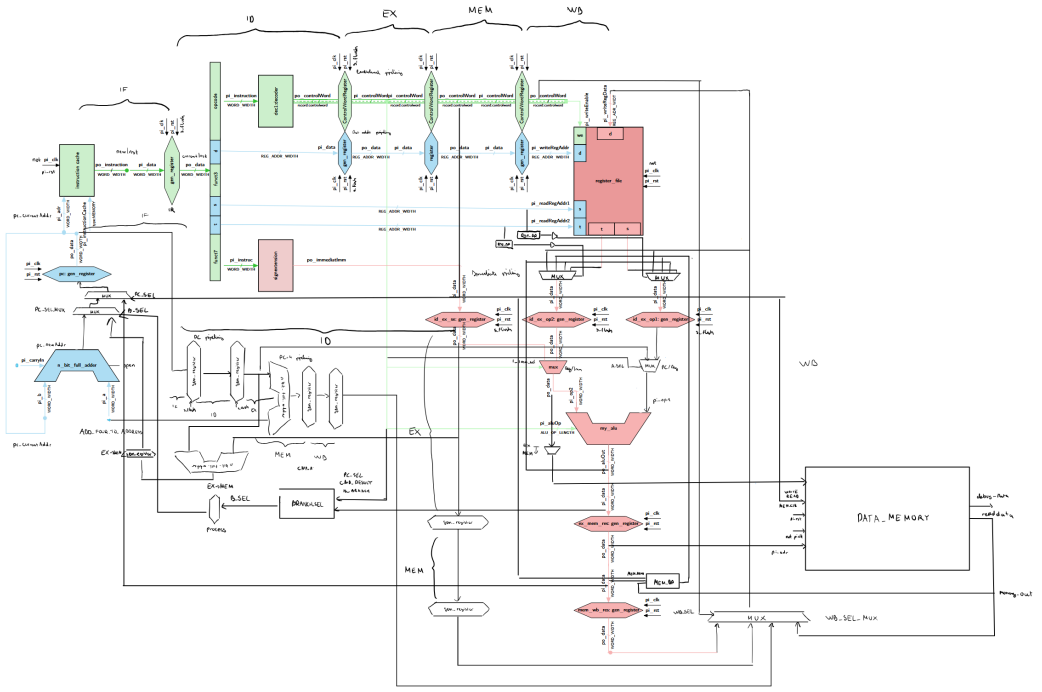

Abbildung 10: RISC-V Aufbau

2.8 Registersatz von RISC-V

Register	Mnemonik	Name	Beschreibung
0	zero	Nullregister	Konstanter Wert: 0
1	ra	Return adress	Rücksprungadresse bei Funktionsaufrufen wie **jal**
2	sp	Stack pointer	Aktueller Stackbereich im Speicher
3	gp	Global pointer	Globale Daten im Datensegment
4	tp	Thread pointer	Datenstruktur des aktuellen Threads
5-7	$t_0 - t_2$	Temporary registers	Temporäre Register
8	s_0 / fp	Saved registers / Frame pointer	Anfang des Stack Frames einer Funktion
9	s_1	Saved register	Inhalt muss vor Überschreiben auf Stack gesichert werden
10-11	$a_0 - a_1$	Function arguments / Return values	Funktionsargumente / Rückgabewerte von Funktionen
12-17	$a_1 - a_7$	Function arguments	Funktionsargumente
18-27	$s_2 - s_{11}$	Saved registers	
28-31	$t_3 - t_6$	Temporary registers	Temporäre Register

Tabelle 6: Registersatz Belegungskonventionen des RV32I Befehlssatzes

2.9 Registersatz der MIPS

Register	Mnemonik	Name	Beschreibung
0	$zero	Nullregister	Konstanter Wert: 0
1	$at	Assembler temporary	Für vom Assembler erzeugte Instruktionen reserviert
2–3	$v0–$v1	Values / Return values	Rückgabewerte von Funktionen
4–7	$a0–$a3	Arguments	Funktionsargumente
8–15	$t0–$t7	Temporaries	Temporäre Register, Aufrufer muss sichern
16–23	$s0–$s7	Saved registers	Muss vom Aufgerufenen bei Benutzung gesichert werden
24–25	$t8–$t9	Temporaries	Weitere temporäre Register
26–27	$k0–$k1	Kernel registers	Nur vom Betriebssystem genutzt
28	$gp	Global pointer	Zeiger auf globale Daten
29	$sp	Stack pointer	Zeiger auf aktuellen Stackbereich
30	$fp	Frame pointer	Basisadresse des Stack Frames
31	$ra	Return address	Rücksprungadresse bei Funktionsaufrufen (z.B. **jal**)

Tabelle 7: Registersatz Belegungskonventionen der MIPS Architektur

3 Speicherarchitektur

3.1 Wahlfreie Speicher

- SRAM bzw. DRAM Zellen werden in einer Speichermatrix angeordnet und per One-Hot-Codierung adressiert. Zur Verbesserung der Laufzeit wird der Speicher in **Bänke** unterteilt.

- Aufgrund der deutlich kleineren Größe von DRAM ($6F^2$) gegenüber

SRAM ($140F^2$) steigt die Größe von Speichermatrizen bei SRAM deutlich schneller an (siehe Abbildung 11), wobei DRAM zusätzlich eine **Refresh Logic** benötigt (daher ist die Schaltungsgröße bei einer kleinen Speichermatrix zuerst größer)

- Zusätzlich wird bei DRAM bei großen Speichermatrizen die Adresse in eine Zeilen-/ (**RAS**) und eine Spaltenadresse (**CAS**) aufgeteilt

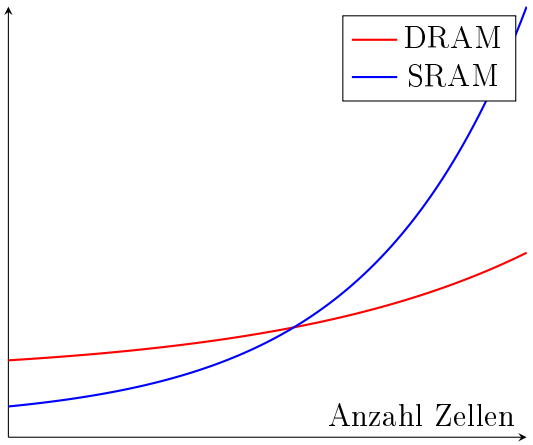

Abbildung 11: Schaltungsgröße bei SRAM vs. DRAM

3.2 Assoziativspeicher / Content-Adressable-Memory (CAM)

- Besteht aus zwei Speichermatrizen, eine SRAM-Matrix zum Ablegen von Daten und einer zweiten Matrix zum Ablegen von Adressen
- Beim Anlegen eines Datums liefert der Assoziativspeicher dann die Adresse der Speicherzelle, in der das Datum abgelegt wurde

3.3 Speicher Optimierung

Definition 3.1: Bank

Eine Bank ist ein unabhängiger Speicherblock innerhalb eines Speichers, bestehend aus vielen Speicherzellen, der separat adressiert und betrieben werden kann. Bei gleichzeitigem Zugriff auf dieselbe Bank kommt es zu einem **Bank-Conflict**, wodurch die Parallelität eingeschränkt wird.

Definition 3.2: Interleaving

Bei der Adressraumverschränkung (Interleaving) wird der Adressraum typischerweise durch **Low-Order-Interleaving**, bei der die niederwertigen Bits der Adresse die jeweilige Bank bestimmen, verschränkt. Dadurch liegen aufeinanderfolgende Adressen in verschiedenen Banks, was parallele Ausführung ermöglicht.

Definition 3.3: Linearer Adressraum

Liste von Speicherwörtern mit monoton steigenden Indizes

Probleme bei Mehrprogrammbetrieb

- Programme können größer sein als Speicher
- Mehrere Programme können im Speicher stehen

Lösungsansätze

- **Programmüberlappung** (Overlay): Auslagerung von Programm auf externen Massenspeicher
- **Paging**: Einteilung des physischen Adressraums in Pages →Hardware zur Adressumsetzung erforderlich (MMU)

3.4 Seitenverwaltung

Definition 3.4: Seite

Eine Seite oder Kachel ist eine Partition eines virtuellen Adressraums mit fester Größe und eindeutiger Seitennummer. Jedes Programm erhält eine Seite, sodass es wirkt, als hätte es einen ganzen Adressraum für sich.

Definition 3.5: Seitenrahmen

Partition eines physischen Speichers, in einen Seitenrahmen passt genau eine Seite des virtuellen Adressraums

Definition 3.6: Overlays

Daten werden dynamisch aus dem Hintergrundspeicher nachgeladen (Paging). Sollten diese nicht im Hauptspeicher liegen, wird eine Unterbrechung (Interrupt / Trap) ausgelöst, sodass das Betriebssystem eine Routine startet, die die entsprechende Seite einlagert. Dabei muss ggf.

eine Seite ersetzt werden, d.h. das Betriebssystem benötigt eine Seitenersetzungsstrategien, die entscheidet, welche Seite ersetzt werden soll.

Definition 3.7: Virtueller Adressraum

Logischer, nicht real vorhandener Adressraum, auf den ein Programm zugreifen kann. Er besteht aus gleich großen Seiten mit jeweils individueller Seitennummer, über diese er adressiert wird.

3.4.1 Seitenersetzungsstrategien

Least Recently Used

- Letzt verwendeter Datenblock wird entfernt
- **Vorteile**: Berücksichtigung zeitlicher Lokalität
- **Nachteile**: Komplex in HW-Umsetzung, hoher Verwaltungsaufwand

First-in First-out

- Längster Block, der im Cache ist, wird entfernt
- **Vorteile**: Einfache implementierung
- **Nachteile**: Nichtbeachtung von Zugriffsverhalten, oft genutzte Daten können früh aussortiert werden

Random

- Zufälliger Block wird erwähnt
- **Vorteile**: Schnell und einfach, keine Verwaltungslogik nötig
- **Nachteile**: Wichtige Daten werden ohne Rücksicht auf Zugriffshäufigkeit aussortiert

Least Frequently Used

- Block mit geringster Zugriffshäufigkeit werden über bestimmen Zeitraum ersetzt
- **Vorteile**: Daten, die häufig benötigt werden, bleiben im Cache
- **Nachteile**: Hoher Verwaltungsaufwand, Früher häufig genutzte Daten werden zu viel Priorität gegeben

3.5 Segmentverwaltung

Definition 3.8: Segmente

Prozesse werden in Segmente, um jedes Programm an einer beliebigen Stelle im Speicher ablegen zu können. Alle Adressangaben sind relativ zu einer *Basisadresse* anzugeben. Der Wert der Basisadresse wird entweder vom *Linker* vorgenommen oder vom Betriebssystem gesetzt. Prozessoren der x86-Familie nutzen ein *Basisregister* für den Start des aktuell auszuführenden Segments bzw. ein *Limitregister*, um die Grenzen der Segmente einzuhalten. Ein Programm kann ein vollständiges Segment ausfüllen oder *dynamisch* auf mehrere aufgeteilt werden.

3.5.1 Einteilung des physischen Speichers

- Prozesse werden in *Code*- bzw. *Textsegment* (statisch) und *Daten*- oder *Stapelsegment* (dynamisch) eingeteilt.
- Stapel (Stack) liegt im oberen Bereich des Speichers (dynamisch nach unten)
- Programm und Konstanten liegen am unteren Ende des Speichers (konstant)
- Darüber werden Variablen gespeichert (dynamisch nach oben)
- Der Raum dazwischen wird Haufen (bzw. Heap) genannt

3.5.2 Virtueller Speicher

- **Memory-Management-Unit** (MMU) ist für die Adressübersetzung und für die Verwaltung der im Hauptspeicher abgelegten Seitentabelle zuständig
- Adressübersetzung läuft häufig mehrstufig, d.h. Einträge einer Seitentabelle verweisen auf weitere Seitentabellen →Auslagerung von Seitentabellen möglich
- **Transaction-Lookaside-Buffer** (TLB) dient als schneller Cache in der MMU für Adressübersetzungen

3.6 Virtuelle Adressen

3.6.1 SV32-Modell

Aufbau

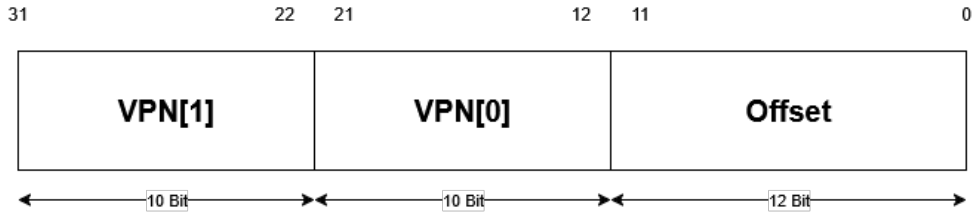

Abbildung 12: Virtuelle Adresse nach dem SV32-Modell

- **VPN[1]**: Index im Root-Page-Table
- **VPN[0]**: Index im Second-Level-Table
- **Offset**: Byte innerhalb der Seite

Adressübersetzung

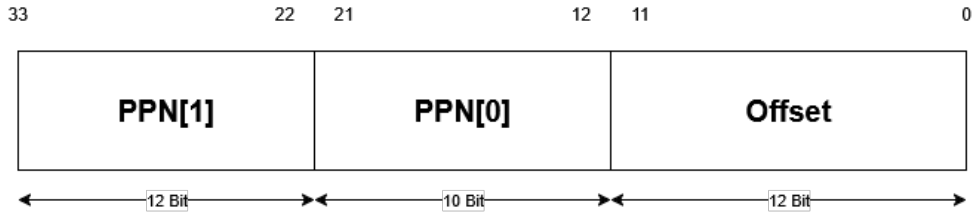

Abbildung 13: Physische Adresse nach dem SV32-Modell

- **SATP-Register** (Supervisor Address Translation and Protection): enthält physische Basisadresse für der Root-Page-Table
- **VPN[1]**: jeder Eintrag (PTE) ist 32 Bit groß, verweist auf Second-Level-Table
- **VPN[0]**: Eintrag im Second-Level-Table enthält physische Basisadresse der Zielseite
- **Offset**: wird direkt übernommen als Offset innerhalb der Seite

3.6.2 MIPS

- Adressübersetzung basierend auf TLB
- Betriebssystem verwaltet Seitentabellen

- Bei TLB-Miss wird ein Interrupt geworfen, der das Betriebssystem anstößt, die Adresse zu übersetzen

3.7 Caches

Definition 3.9: Block

Ein Cache-Block bzw. eine Cache-Line ist die kleinste Dateneinheit, die zwischen dem Hauptspeicher und dem Cache verschoben wird. Wird eine Speicheradresse angefragt, wird der gesamte Block, in dem das Wort steht, verschoben. Ein Cache-Block besteht aus

1. **Tag** bzw. Etikett: Bezeichner für Speicherblock
2. **Datenfeld**: Nutzdaten aus dem RAM
3. **Statusbits**:
    - **Valid-Bit**: Eintrag im Cache stimmt mit Hauptspeicher überein
    - **Dirty-Bit**: Hauptspeicher wird nicht gleichzeitig mit Cache aktualisiert

Definition 3.10: Set

Ein Cache-Set ist eine Gruppe von Blöcken, in die ein bestimmter Speicherbereich abgebildet wird. Beim Anfragen einer Adresse aus dem RAM wird diese in drei Teile zerlegt:

1. **Tag**: liegt Block im Cache?
2. **Index**: Identifikation des Sets
3. **Block-Offset**: Wort innerhalb des Blocks

Definition 3.11: Lokalitätsprinzip

Das Lokalitätsprinzip spielt eine wichtige Rolle beim Caching, aber auch z.B. in der Seitenverwaltung (3.4)

- **Zeitliche Lokalität**: bereits verwendete Daten werden wahrscheinlich bald wieder benötigt
- **Räumliche Lokalität**: beim Verwenden einer Speicherstelle ist die Wahrscheinlichkeit hoch, dass eine benachbarte ebenfalls bald

benötigt wird

- Beim Zugriff auf Speicherwörter werden diese häufig erneut bzw. benachbarte Speicherwörter angesprochen.
- *Bus Snooping*: Beim Lese-/ Schreibzugriff auf Hauptspeicher wird Cache aktiviert
- Einteilung in *Tagspeicher* & *Datenspeicher*, realisiert durch Assoziativspeicher / Content-Adressable-Memory (CAM)
- Stimmt die Blocknummer aus dem Adresswort des Prozessors und die im Tagspeicher überein, wird ein *Cache Hit* signalisiert
- Caches sind räumlich stark begrenzt →nicht jeder Eintrag kann gecached werden, es kommt zu *Cache Misses*
- *Cache-Line* besteht aus mehreren Speicherwörtern, die zu einem *Block* zusammengefasst werden

3.7.1 Cache-Assoziativität

Definition 3.12: Assoziativität

Freiheitsgrad bei der Platzierung von Speicherblöcken im Cache

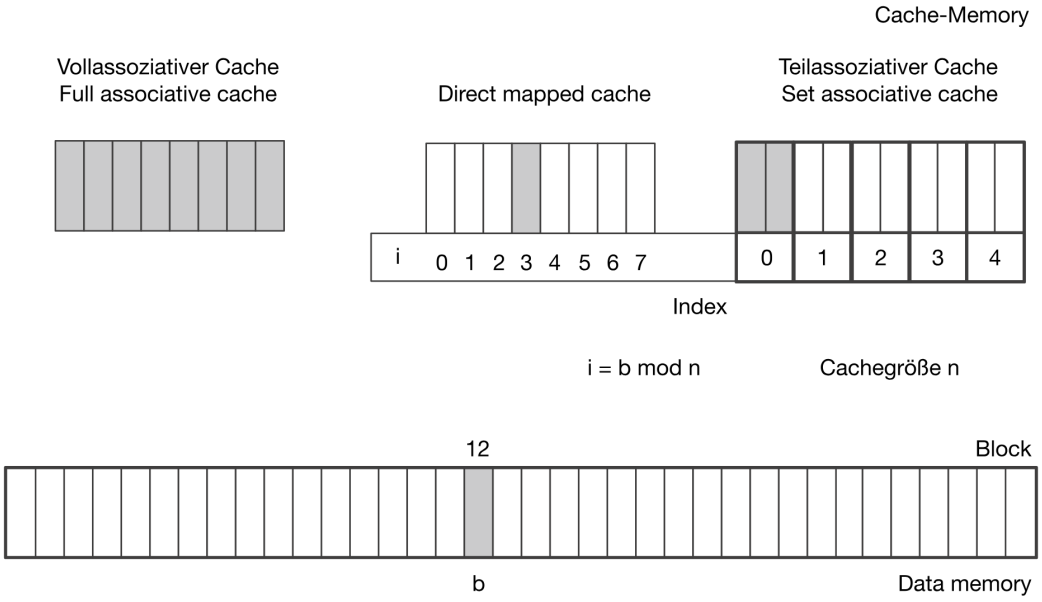

Abbildung 14: Abbildung des Hauptspeichers durch den Cache

Direct-mapped-Cache (direkt-abgebildet)

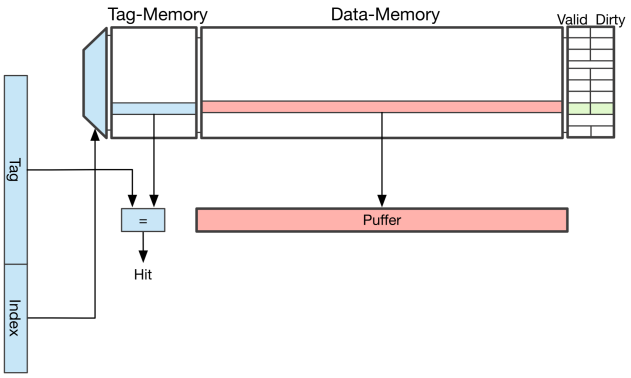

Abbildung 15: Direct-mapped Cache

- Jeder Block hat eindeutig zugewiesenen Platz im Cache z.B. $i = b \mod m$, wobei i = Index, b = Blocknummer, m = Gesamtanzahl Blöcke
- Assoziativität = 1, d.h. 1-fach-assoziativ
- **Adressformat**: [Tag | Index | Block Offset]
- **Vorteil**: Mit wenig Hardwareaufwand realisierbar, schnelle Zugriffszeiten

- **Nachteil**: Viele Kollisionen möglich, wenn mehrere Daten auf denselben Cache-Block abgebildet werden

Vollassoziativer Cache (fully associative)

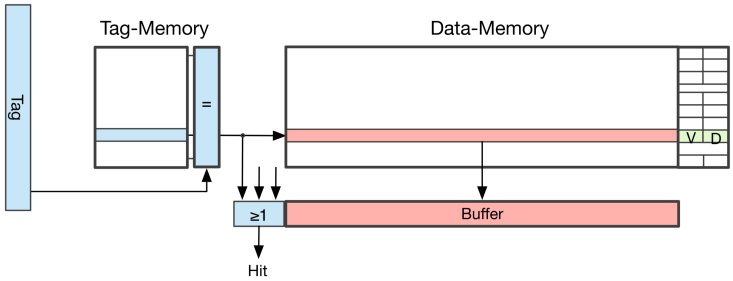

Abbildung 16: Vollassoziativer Cache

- Jeder Block kann an jeder Stelle im Cache gespeichert werde.
- Kein fester Index →gesamte Adresse (bzw. Tag) wird zum Vergleich herangezogen
- **Adressformat**: [Tag | Block Offset] (kein Index, da jeder Block in jede Zeile gelegt werden kann)
- **Vorteil**: Maximale Flexibilität, minimale Konflikte.
- **Nachteil:** Hoher Hardwareaufwand, muss alle Cache-Einträge parallel vergleichen

Teilassoziativer bzw. Set-assoziativer Cache (n-fach assoziativ)

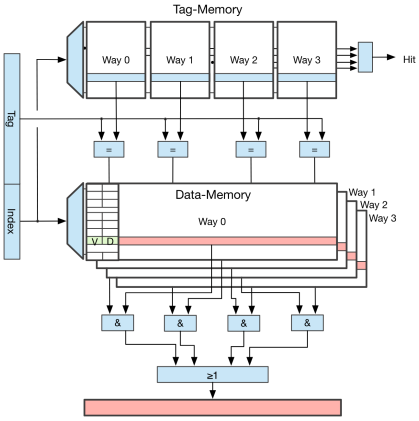

Abbildung 17: 4-fach assoziativer Cache

- Speicherblock kann in n verschiedenen Blöcken innerhalb eines Sets gespeichert werden
- Jedes Set enthält genau n Blöcke
- Cache ist in Sets unterteilt; jeder Set enthält mehrere Lines (Ways)
- **Adressabbildung**:
  - Index →wählt den Set (z.B. Adresse *mod* Sets)
  - Innerhalb des Sets: beliebige Line (assoziativ)
- **Adressformat**: [Tag | Set Index | Block Offset]
- **Kompromiss**: weniger Konflikte als Direct-Mapped, günstiger als Fully Associative

Eigenschaft	**Direct-mapped**	**Vollassoziativ**
Zugriffszeit	schnell	langsam
Komplexität	gering	hoch
Konfliktrate	hoch	gering
Hardware-Aufwand	gering	hoch

Tabelle 8: Vergleich von direct-mapped und vollassoziativen Caches

3.7.2 Ersetzungsstrategien

Least Recently Used

- Letzt verwendeter Datenblock wird entfernt
- **Vorteile**: Berücksichtigung zeitlicher Lokalität
- **Nachteile**: Komplex in HW-Umsetzung, hoher Verwaltungsaufwand

First-in First-out

- Längster Block, der im Cache ist, wird entfernt
- **Vorteile**: Einfache implementierung
- **Nachteile**: Nichtbeachtung von Zugriffsverhalten, oft genutzte Daten können früh aussortiert werden

Random

- Zufälliger Block wird erwähnt
- **Vorteile**: Schnell und einfach, keine Verwaltungslogik nötig
- **Nachteile**: Wichtige Daten werden ohne Rücksicht auf Zugriffshäufigkeit aussortiert

Least Frequently Used

- Block mit geringster Zugriffshäufigkeit werden über bestimmen Zeitraum ersetzt
- **Vorteile**: Daten, die häufig benötigt werden, bleiben im Cache
- **Nachteile**: Hoher Verwaltungsaufwand, Früher häufig genutzte Daten werden zu viel Priorität gegeben

3.7.3 Write Allocation

Write-Allocate (Fetch-on-Write)

- Block wird aus dem Hauptspeicher zuerst in den Cache geladen
- **Dirty-Bit** im Cache
  - **Dirty**: Block wurde verändert, beim Verdrängen in Hauptspeicher zurückschreiben
  - **Clean**: Block unverändert, kein Zurückschreiben nötig
- **Write-Back**: Hauptspeicher wird erst dann aktualisiert, wenn Cacheblock verdrängt wird
  - **Vorteil**: Sehr effizient, da viele Writes nur im Cache
  - **Nachteil**: Komplex, Gefahr von Inkonsistenzen →Cache-Kohärenz notwendig

No-Write-Allocate (Write-No-Allocate / Write-Around)

- Block wird nicht in den Cache geladen
- Schreibzugriff direkt auf Hauptspeicher
- **Write-Through**: Schreibzugriff wird sofort in Cache und Hauptspeicher geschrieben

- **Vorteil**: Hauptspeicher konsistent
- **Nachteil**: Langsam, weil jeder Schreibzugriff auch auf Hauptspeicher zugreift

3.7.4 MESI-Protokoll

- Cache-Kohärenzprotokoll, das sicherstellt, dass in einem Mehrkernsystem alle Prozessorkerne konsistente Daten in ihren Caches haben
- **M(odified)**
  - Cache-Line wurde verändert und ist exkl. im lokalen Cache vorhanden
  - Hauptspeicher veraltet →Write-Back bei Speicherfreigabe
- **E(xclusive)**
  - Zeile ist identisch zum Hauptspeicher und exkl. im lokalen Cache vorhanden
  - Lesender Zugriff ist erlaubt, Schreiben macht daraus "Modified"
- **S(hared)**
  - Zeile im Hauptspeicher ist gültig, wird von mehreren Caches gleichzeitig gehalten
  - Nur Lesezugriffe erlaubt, Schreibzugriffe führt zu Invalidation bei anderen
- **I(nvalid)**
  - Cache-Line ist ungültig und muss neu geladen werden
  - Bei jeder Änderung einer Cache-Line →Benachrichtigung anderer Caches zur Verhinderung veralteter Daten

3.7.5 Translation Lookaside Buffer (TLB)

- Kleiner assoziativer Cache in der Memory-Management-Unit (MMU) der CPU
- Caching von Adressumsetzungen, da viele Zugriffe auf Seitentabellen ineffizient

- Bei TLB-Miss wird ein Interrupt-Signal ans Betriebssystem geschickt, dass den TLB-Eintrag nachlädt und die Instruktion wiederholt

Unterschiede zwischen RISC-V und MIPS

- **RISC-V**: flexible MMU-Modelle, häufig softwareverwaltete TLB
- **MIPS**: hardwareverwaltete TLB mit festen Assoziativitätsstrukturen

3.8 Uniform vs. Non-Uniform Address Space

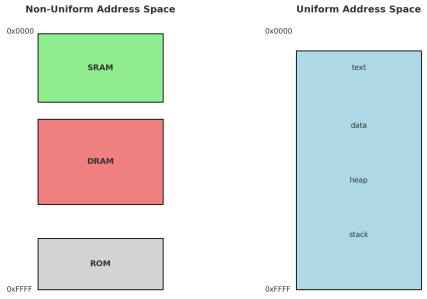

Abbildung 18: Uniform vs. Non-Uniform Address Space

3.8.1 Uniform Adress Space

- Speicheradressierung für alle Adressen gleich
- Keine Unterschiede zwischen Adressen verschiedener Speicherbereiche
- Klassische Von-Neumann-Architektur
- **Vorteile**: Einfache Programmierung
- **Nachteile**: Schlechtere Skalierbarkeit, da alle Zugriffe gleich behandelt werden müssen

3.8.2 Non-Uniform Adress Space

- Zugriffszeiten variieren
- Unterschiedliche Adressräume (Getrennte Bereiche für Code, Daten und I/O)

- Klassische Harvard-Architektur
- **Vorteile**: Hohe Leistung durch Nähe von Speicher und Recheneinheit
- **Nachteile**: Programmierer muss darauf achten, wo Daten liegen (Weniger Transparenz)

3.9 Speicherhierarchie

Zugriffszeit hängt von Länge der Leitungen und somit der Größe des Adressraums ab, der Speicher wird daher zur Erhöhung der Geschwindigkeit virtuell aufgebrochen

Speicher	Aufbau	Geschwindigkeit
Register	Flip-Flops	Sehr schnell
Caches (z.B. L1/L2/L3)	SRAM-Zellen	Schnell
Hauptspeicher	DRAM-Zellen	Langsam
Hintergrundspeicher	NAND-Flash, Magnet-Scheiben	Sehr langsam

Tabelle 9: Speicherhierarchie nach Geschwindigkeit

4 Mikroarchitektur

Definition 4.1: Mikroarchitektur

Die Mikroarchitektur beschreibt den inneren Aufbau eines Rechners, insbesondere die Struktur von Steuer- und Rechenwerk. Während die Architektur festlegt, was die Maschine tun soll, definiert die Mikroarchitektur, wie dies umgesetzt wird. Sie implementiert die Spezifikation der Architektur mithilfe des Mikroprogramms. Bei CISC-Rechnern umfasst sie typischerweise ein Steuerwerk und ein Rechenwerk (Data Path), während bei RISC-Rechnern die Trennung weniger deutlich ist, da sie für skalierbare bzw. superskalare Fließbandverarbeitung optimiert sind.

4.1 Grundlegende Architekturtypen

4.1.1 Von-Neumann/Princeton-Architektur

- Daten und Befehle im selben Speicher

- Befehlszyklen: **IF** (Instruction Fetch), **ID** (Instruction Decode), **OF** (Operand Fetch), **EX** (Execute), **OS** (Operand Set)
- **Vorteil**: kompakter, kosteneffizienter Aufbau
- **Nachteil**: **Von-Neumann Flaschenhals**: gleichzeitiger Zugriff auf Daten und Befehle nicht möglich

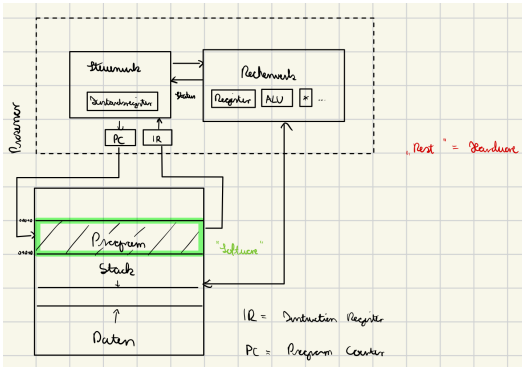

Abbildung 19: Von Neumann Architektur

4.1.2 Harvard-Architektur

- Daten und Befehle in zwei unterschiedlichen Speichern
- Befehlszyklen: **IF** (Instruction Fetch), **ID/OF** (Instruction Decode + Operand Fetch), **EX** (Execute), **MEM** (Memory), **WB** (Write-Back)
- **Vorteil**: hohe Parallelität
- **Nachteil**: aufwändig und teuer in Hardware

4.1.3 Gemischte Architektur

- Nach außen gemeinsamer Speicher, intern separate Caches (Instruktions-Cache ud Daten-Cache) für Befehle und Daten
- Flexible und kosteneffiziente Lösung

4.2 Steuerwerk

Definition 4.2: Steuerfluss

Ablauf von Steuersignalen im Prozessor, mit denen das Steuerwerk die Ausführung der einzelnen Instruktionen in den Funktionseinheiten koordiniert

Definition 4.3: Mikroprogramm

Sequenz von **Mikroinstruktionen**, die zusammen die Steuerung der Hardware für die Ausführung eines Maschinenbefehls übernehmen. Das Mikroprogrammsteuerwerk eines Rechners wird durch ein speicherprogrammierbares PLA realisiert. Ist dieser Speicher als ROM (Read-Only-Memory) realisiert, spricht man von einer **statischen Mikroprogrammierung**, bei einem RAM (Random-Access-Memory) von **dynamischer Mikroprogrammierung**.

4.2.1 Vertikale vs. Horizontale Mikroprogrammierung

- **Vertikale Mikroprogrammierung**: Mikroprogramm ist in viele sequenzielle Schritte unterteilt
  - **Vorteil**: Kürze Mikrobefehle →spart Speicherplatz und komplexe Dekodierlogik
  - **Nachteil**: Langsamer durch mehr Dekodierstufen und weniger Parallelität durch sequenzielle Ausführung
- **Horizontale Mikroprogrammierung**: Maschinen, die einen Befehl in einem Befehlszyklus abarbeiten, lassen sich mit jeweils einem Steuerwort pro Befehlszyklus steuern
  - **Vorteil**: Schnelle Ausführung, da keine weitere Dekodierung notwendig, hohe Parallelität und mehr Flexibilität
  - **Nachteil**: Mehr Speicherbedarf durch breitere Mikrobefehle, aufwändige Steuerung und komplexere Mikroprogramme

4.2.2 Mikroprogrammsteuerwerk

- Ein Mikroprogrammsteuerwerk wird häufig in CISC-Maschinen benötigt, da sich die komplexen Befehlen häufig nicht direkt mit festverdrahteter Logik abbilden lässt
- **Aufbau**

1. **Mikroprogrammspeicher** (Control Store) enthält Mikroprogramme für Maschinenbefehle
2. **Mikroprogrammzähler** (μPC): zeigt auf aktuelle Mikroinstruktion im Mikroprogrammspeicher
3. **Mikroinstruktionen**: Vektor von Steuersignalen, die direkt die Hardware ansteuern
4. **Mikroinstruktionsdecoder**: wandelt Mikroinstruktion in konkrete Steuerimpulse um

4.2.3 Control Hazards

Definition 4.4: Steuerflussabhängigkeit

Bis das Ergebnis eines Sprungergebnis feststeht, ist nicht klar, welche Instruktionen als nächstes geladen werden sollen

Lösungen

- **Stalling**: Pipeline anhalten, bis Sprungergebnis bekannt ist
- **Branch Prediction**: bei bereits genommenen Branches ist die Wahrscheinlichkeit hoch, dass dieser wieder genommen wird
- **Delayed Branch**: Compiler nützt Lücke vor dem Branch

4.2.4 Branch Prediction

Definition 4.5: Kontrollfluss

Logische Abfolge von Befehlen innerhalb eines Programms

- **Ziel**: Vorhersage des Kontrollfluss, s.d. Pipeline nicht auf Auswertung der Bedingung warten muss →weniger Stalls (Wartezyklen)
- **Strategien**
  - **Statisch**: feste Regeln für Vorhersagen
  - **Dynamisch**: Vorhersage berücksichtigt Verhalten vergangener Sprünge
- **Fehlvorhersage** (Misprediction): Pipeline hat falsche Befehle vorgeladen, diese müssen verworfen (geflusht) werden →Geringere Leistung durch Pipeline-Bubbles

- Besonders bei tiefen Pipelines erzeugen Fehlvorhersagen hohe Leistungsverluste, da mehrere Stufen geflusht werden müssen

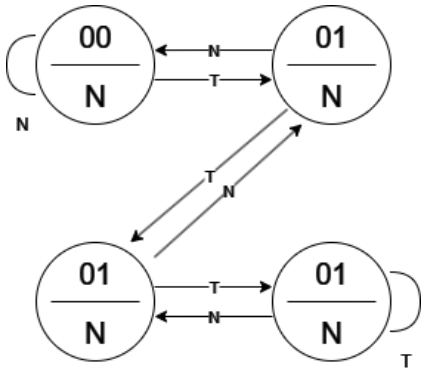

Abbildung 20: Einfacher Branch Prediction Automat

4.3 Rechenwerk

4.3.1 Vektorielles Rechenwerk

- **MAC**-Einheit (Multiply And Accumulate) führt in Hardware die Rechnung $a = a + (b \cdot c)$ in einem Taktzyklus durch
- Ein Rechenwerk mit mehreren parallelen MAC-Einheiten kann so effizient Vektor- und Matrizenoperationen durchführen
- Typische Operation: **SAXPY**

4.3.2 Skalares Rechenwerk

Definition 4.6: Skalares Rechenwerk

Durch Überlappen der Befehlsphasen lässt sich der Datendurchsatz deutlich erhöhen

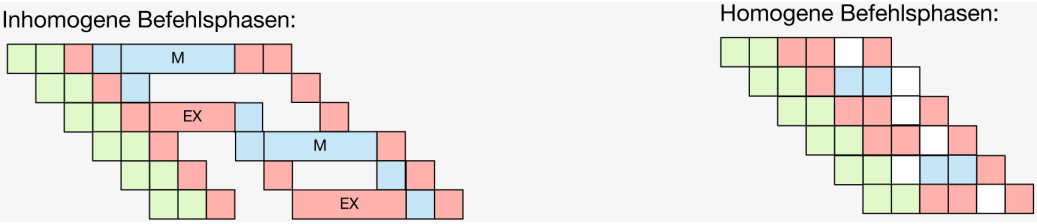

Abbildung 21: Homogone vs. Inhomogene Befehlsphasen

Vorteile von homogenen Befehlsphasen

- Vollständige Überlappung der Befehlsphasen möglich →Befehlsdurchsatz kann deutlich erhöht werden
- **Problem**: Zugriff auf Speicher muss gleich schnell sein wie Bearbeitung anderer Befehlsphasen

4.3.3 Fließbandverarbeitung

- **Problem**: Durch Homogene Befehlsphasen (Abbildung 21) kommt es zu gleichzeitigem Zugriff auf den Speicher (**OF** und **OS**) bei der **Von-Neumann-Architektur** (Von-Neumann-Flaschenhals)
- **Lösung**: 3-Adressmaschine (Abbildung 4) mit gemischter **Harvard-Architektur** (eigener Instruktions-Cache) und Load-Store-Architektur

4.3.4 Data Hazards

Definition 4.7: Datenabhängigkeit

Eine Instruktion benötigt ein Ergebnis, das eine frühere Instruktion erst berechnen muss →**RAW**-Hazard (Read-After-Write)

```
1  I1: R1 = R2 + R3
2  I2: R4 = R1 + 5 ; braucht Ergebnis von I1
```

Definition 4.8: Gegenabhängigkeit

Eine spätere Instruktion möchte in ein Register schreiben, das vorher noch von einer früheren Instruktion gelesen wird →**WAR**-Hazard (Write-after-Read), tritt selten in RISC-Pipelines auf, da Leseoperation (ID) früh und Schreiboperation (WB) spät

```
I1: R4 = R2 + R3   ; liest R2
I2: R2 = R5 + 1    ; schreibt R2
```

Definition 4.9: Ausgabeabhängigkeit

Reihenfolge der Schreiboperationen könnte vertauscht werden →**WAW**-Hazard (Write-after-Write), tritt nur bei *Out-of-Order Execution* auf

```
I1: R1 = R2 + 5
I2: R1 = R3 + 7
```

Lösungen

- **Interlocking**: Erkennung von Datenabhängigkeiten und automatisches Einfügen von Pipeline-Stalls (NOPs), bis Daten oder Ressource verfügbar
- **Hardware Forwarding / Bypassing**: Datenpfade leiten Ergebnisse aus vorherigen Berechnungen an EX-Stufe weiter
- **Compiler-Techniken**: Compiler fügt gezielt NOPs ein oder verschiebt unabhängige Instruktionen dazwischen

4.3.5 Structural Hazards

- Treten auf, wenn zwei Pipeline-Stufen gleichzeitig auf dieselbe Hardware-Ressource zugreifen wollen
- **Lösung**: getrennter Instruktions- und Datencache, Stalling bei blockierter Ressource

4.4 Out-Of-Order Execution

4.4.1 Scoreboarding

- **Scoreboard**: zentrale Kontrolllogik, die Funktionseinheiten (FUs) und Register überwacht, um Data Hazards zu vermeiden
- **Ablauf in Phasen**
  1. **Issue**: Prüfung einer Instruktion

- Ist die FU frei?
- Liegt Hazard-Bedingung vor?

2. **Read Operands**: Sobald Operanden frei sind, darf die Instruktion sie lesen
3. **Execute**: Instruktion wird in der FU ausgeführt
4. **Write Result**: Ergebnis wird erst zurückgeschrieben, wenn kein WAW/WAR-Hazard entsteht

4.4.2 Tomasulo-Algorithmus

Definition 4.10: Functional Unit

Eine Functional Unit (FU) ist ein Prozessorbaustein, der logisch/arithmetische Berechnungen ausführt. FUs unterscheiden sich in der Art der Operationen, welche sie ausführen können (z.B. ADD, MUL LOAD/-STORE, usw.)

Definition 4.11: Reservation Station

Jeder FU sind meist mehrere Reservation Stations (RSs) zugeordnet, die **Register-Renaming** implementieren und wie temporäre Register verwendet werden

Definition 4.12: Common Data Bus

Ein Common Data Bus (CDB) dient zum Austausch von Ergebnissen zwischen den RSs

Funktionsweise

1. **Issue**
   - Befehl → freie Reservation Station
   - Operanden sofort übernehmen (falls gültig) oder Tag der produzierenden RS eintragen
   - Wenn keine RS frei → Warten in der Queue
2. **Execute**
   - Start, sobald alle Operanden da sind
   - Falls Operanden fehlen → CDB überwachen und Werte nachladen
3. **Write Result**

- Ergebnis + Tag der RS auf den CDB legen
- Alle wartenden RS und die Registerdatei aktualisieren

4.5 CISC vs. RISC

CISC

- Komplexe Instruktionen mit variabler Länge (siehe 2.6)
- Instruktionen werden in Micro-Ops (μOps) zerlegt
- Pipeline ist tief und unregelmäßig
- Komplexe Hazard Erkennung notwendig (Out-of-Order-Scheduling, Register Renaming, Speculative Execution)

RISC

- Feste Instruktionslänge, Load-Store-Architektur
- Einfache Überlappung der Befehlsphasen möglich
- Hazard-Behandlung per Forwarding, Branch Prediction

4.6 Rechenleistung

4.6.1 Gatterlaufzeit

Ein **CMOS**-Gatter bzw. ein RC-Glied lässt einfach als **Schalter + Widerstand + Kondensator** modellieren.

$$\textbf{Ladevorgang} \quad U(t) = U_0 \cdot (1 - e^{-\frac{t}{R \cdot C}})$$

$$\textbf{Entladevorgang} \quad U(t) = U_0 \cdot e^{-\frac{t}{R \cdot C}}$$

Ein guter Näherungswert für die Gatterlaufzeit ist durch Abschätzung und Vernachlässigung der Induktivität:

$$t = R \cdot C$$

4.6.2 Prozessortakt

Definition 4.13: Prozessortakt

Der maximale Prozessortakt lässt sich wie folgt berechnen:

$$f_{max} \approx \frac{1}{N \cdot t}$$

wobei N die Länge die **kritischen Pfadlänge** ist (längster Pfad an hintereinander geschalteten Gattern) und t die Gatterlaufzeit ist. Die Zykluszeit (Zeit für die Abarbeitung eines Befehls) berechnet sich wie folgt:

$$T = \frac{1}{f}$$

4.6.3 Technologische Skalierung

Zur Verbesserung des Prozessortakt gilt es daher, die beiden Faktoren R und C möglichst gering zu halten.

$$\textbf{Elektrischer Widerstand} \quad R = \rho \cdot \frac{l}{A}$$

wobei ρ der spezifische Widerstand bzw. die Materialkonstante ist, l die Länge und A die Querschnittsfläche der Leitung ist.

$$C = \mathcal{E}_0 \cdot \mathcal{E}_r \cdot \frac{A}{d}$$

Definition 4.14: Mooresches Gesetz

Die Anzahl an Transistoren auf einem integrierten Schaltkreis wird in etwa alle 18-24 Monate verdoppelt.

4.6.4 Performance-Steigerung durch Architektur (Iron Law)

$$p := n_p \cdot C_{PI} \cdot T_C$$

wobei p die Performance eines Rechnersystems darstellt, n_p die Anzahl an Befehlen, C_{PI} die Anzahl an Zyklen pro Instruktion und T_C die Befehlszykluszeit.

5 Assembler-Programmierung

5.1 Label

Definition 5.1: Label

Ein Label dient zur Markierung von Speicheradressen für Code und Daten

- **Globale Labels** durch Direktive .globl
- Assembler ersetzt Labels beim Übersetzen durch konkrete Adressen oder Offsets
- Programmteile können verschoben werden, ohne Sprungadressen manuell anzupassen

5.2 Direktive

Definition 5.2: Direktive

Definition von Daten, Speicher, Symbole oder Struktur für den Assembler

- **Speicher**
  - **.word**: eine bzw. mehrere 32-Bit Zahlen
  - **.byte**: eine bzw. mehrere Bytes
  - **.asicz**: String + Nullterminator
- **Sektionen**
  - **.text**: enthält Programmcode
  - **.data**: definiert Daten
  - **.globl main**: markiert Label main als Startpunkt des Programms

5.3 Makros

- Erstellung benutzerdefinierter Befehle, die in der ISA (Instruction Set Architecture) nicht definiert sind
- Assembler ersetzt Befehle automatisch in einen oder mehrere MIPS-Instruktionen

```
.macro square reg
  mul \reg, \reg, \reg
.endm       # RISC-V
.end_macro  # MIPS
```

5.4 RISC-V vs. MIPS Assembler

Unterschied	RISC-V	MIPS
Notation von Registern	Präfix x	Präfix $
Registersatz	Siehe Tabelle 6	Siehe Tabelle 7

Tabelle 10: Unterschiede zwischen der ISA der RISC-V und der MIPS

5.5 Unterprogrammaufrufe

- **Caller-saved** (volatile): Aufrufer sichert Register auf Stapel
  - **Übergabeparameter**: $a_7 - a_2$
  - **Temporäre Werte**: $t_6 - t_0$
  - **Rückgabewerte**: $a_1 - a_0$
- **Callee-saved** (non-volatile): Aufgerufener sichert Register
  - **Rücksprungadresse**: ra
  - **Rahmenzeiger**: fp / tp / gp
  - **Sicherungsregister**: $s_{11} - s_0$

5.6 Beispiele

5.6.1 Rekursive Fakultäts-Funktion

```
.data
n: .word 5
result: .word 0

.text
.globl main

main:
  lui t0, %hi(n)
```

```
10      lw a0, %lo(n)(t0)
11
12      # Function call
13      jal ra, fak
14
15      lui t0, %hi(result)
16      sw a0, %lo(result)(t0)
17
18      j end
19
20    fak:
21      # Save return adress and return value to stack
22      addi sp, sp, -8
23      sw ra, 4(sp)
24      sw a0, 0(sp)
25
26      # Base case (n <= 1)
27      li t0, 1
28      ble a0, t0, base_case
29
30      # Recursive function call
31      addi a0, a0, -1
32      jal ra, fak
33
34      # Multiply results
35      lw t1, 0(sp)
36      mul a0, a0, t1
37
38      j fak_end
39
40    base_case:
41      li a0, 1
42
43    fak_end:
44      lw ra, 4(sp)
45      addi sp, sp, 8
46      jr ra
47
48    end:
49      nop
```

5.6.2 Bubble-Sort

```
.data
array:  .word 5,3,4,1,2
length: .word 5

.text
.globl main

main:
  lw      t0, length
  addi    t0, t0, -1

outer_loop:
  beq     t0, zero, done
  li      t1, 0

inner_loop:
  # End of loop
  lw      t2, length
  addi    t2, t2, -1
  sub     t3, t2, t1
  beq     t3, zero, end_inner

  # Address calculation
  slli    t4, t1, 2
  la      t5, array
  add     t6, t5, t4

  # Load both values
  lw      a1, 0(t6)
  lw      a2, 4(t6)

  blt     a1, a2, no_swap

  # Swap
  sw      a2, 0(t6)
  sw      a1, 4(t6)

no_swap:
  addi    t1, t1, 1
  j       inner_loop

end_inner:
  addi    t0, t0, -1
```

```
44    j       outer_loop
45
46  done:
47    nop
```

6 Programmieren in Hochsprache

6.1 Übersetzungsprozess einer Hochsprache

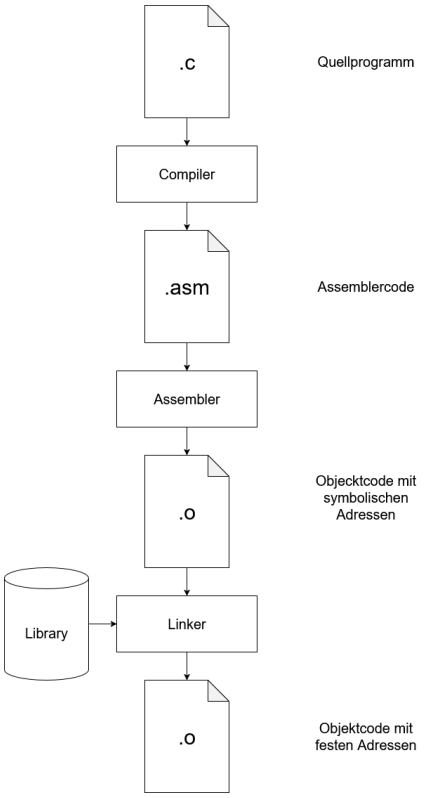

Abbildung 22: C Übersetzungsprozess

- Hochsprachen wie Java oder C++ werden in Textdatei gespeichert
- **Compiler** erzeugt daraus Assembler-Code
- **Assembler** übersetzt schlussendlich Assembler-Code in Maschinen- bzw. Objektcode

- **Linker** ist ein Hilfsprogramm des Assemblers und erlaubt, Programm beliebig im Speicher abzulegen und das Maschinenprogramm mit wiederverwendbaren Bibliotheken zu verknüpfen

6.2 Schablonen

Definition 6.1: Grundblock

Ein Grundblock bzw. ein Basisblock ist eine Folge von Anweisungen zwischen zwei Verzweigungen. Er enthält selbst keine bedingten oder unbedingten Sprungbefehle.

Definition 6.2: Steuerblock

Bestimmt anhand einer Bedingung die nächste Verzweigung. In einem Grundblockgraph stellt er einen Knoten dar.

Definition 6.3: Grundblockgraph

Gerichteter bipartiter Graph, der den Kontrollfluss eines Programms darstellt.

6.2.1 Verzweigung

```
# Hochsprache
if (Steuerblock) {
  BasisBlockTrue
} else {
  BasisBlockFalse
}

# Assembler
cb:   beq $t1, $t2, if
if:   BBTrue
      j endif
else: BBFalse
endif: ...
```

6.2.2 for-Schleife

```
# Hochsprache
for (i=0; i<n; i++) {
  BasisBlock
}
```

```
5
6   # Assembler
7   for: addiu $t1, $t1, $zero # n=0
8        sltu $t3, $t1, $t2    # Wenn i<n: $t3=1
9        addiu $t1, $t1, 1     # n++
10       BB                    # BasisBlock
11       bne $t3, $zero, for   # Wenn $t3 != 0 spring zu for
```

6.2.3 while-Schleife

```
1   # Hochsprache
2   while (i++ < m) {
3     BB
4   }
5
6   # Assembler
7   while: sltu $t3, $t1, $t2      # Wenn i<n: $t3 = 1
8          addiu $t1, $t1, 1       # n++
9          BB                      # BasisBlock
10         bne $t3, $zero, while   # Wenn $t3 != 0 springe zu while
```

6.2.4 do-Schleife

```
1   # Hochsprache
2   do {
3     BB
4   } while(i++ < n)
5
6   # Assembler
7   do:
8     BB                  # BasisBlock
9     sltu $t3, $t1, $t2  # Wenn i<n: $t3 = 1
10    addiu $t1, $t1, 1   # n++
11    bne $t3, $zero, do  # Wenn $t3 != 0 springe zu do
```

6.3 Programme in C

6.3.1 Rekursive n-te Fibonacci-Zahl

```
1 int rFib(int n) {
2   switch(n) {
3     case 0: return 0;
4     case 1: return 1;
```

```
    default: return rFib(n-1) + rFib(n-2);
  }
}
```

6.3.2 Iterative n-te Fibonacci-Zahl

```
int iFib(int n) {
  int i = 2;
  int F, n_0 = 0;
  int n_1 = 1;

  do {
    if (n<1) F=0;
    else {
      F = n_0 + n_1;
      n_0 = n_1;
      n_1 = F;
    }
  } while (i++ < n);
  return F;
}
```

7 Hardware Simulation mit VHDL

7.1 Grundlegende Datentypen

7.1.1 Eingebaut (STD.STANDARD)

- **boolean**: true | false
- **bit**: '0' | '1'
- **integer**: ganze Zahlen
- **real**: Gleitkommazahlen
- **time**: Zeit (fs, ps, ns, us, ms, sec, min, hr)
- **character**: einzelnes Zeichen
- **string**: Array aus character
- **enumeration**: Aufzählungstypen z.B. type state is (IDLE, BUSY, DONE);

- **array**: beliebige Felder z.B. type mem is array (0 to 255) of bit;
- **record**: Datensätze

7.1.2 IEEE-Pakete

IEEE.STD_LOGIC_1164

- **std_ulogic**
- **std_logic**: U, X, 0, 1, Z, W, L, H

IEEE.NUMERIC_STD

- **signed**: vorzeichenbehaftet
- **unsigned**: nicht vorzeichenbehaftet

7.1.3 Signale vs. Variablen

- **signal**: Simulation von Leitungen, mit der sich Schaltkreise verbinden lassen
- **variable**: speichert Datentypen, nur in deklarativen Teilen (process, procedure, function) verwendbar